THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Institut National Polytechnique de Toulouse (INP Toulouse)*

Présentée et soutenue le *jj/mm/2021* **par :**

Daniel LOCHE

Prévention des fautes temporelles sur architectures multicœur pour les systèmes à criticité mixte

---

JURY

Premier MEMBRE	Professeur d'Université	Président du Jury
Jean-Charles FABRE	Toulouse INP	Membre du Jury
Troisième MEMBRE	Chargé de Recherche	Membre du Jury
Michael LAUER	Université Toulouse 3	Membre du Jury
Cinquième MEMBRE	Chargé de Recherche	Membre du Jury

École doctorale et spécialité :
EDSYS : Systèmes embarqués 4200046
Unité de Recherche :
Laboratoire d'analyse et d'architecture des systèmes
Directeur(s) de Thèse :
Jean-Charles FABRE et *Michael LAUER*
Rapporteurs :
Premier RAPPORTEUR et *Second RAPPORTEUR*

Remerciements

A faire en dernier :-)

Table des matières

Introduction

0.1 Systèmes embarqués automobiles

0.1.1 Évolutions des systèmes embarqués

Système mécanique => système cyber-physique

0.1.2 Architectures EE

=> Augmentation complexité architecture EE => augmentation des besoins (puissance de calcul, ADAS, voiture connectée/autonome...)

0.2 Tendances et Contraintes actuelles

0.2.1 Tendances

=> Nouvelles architectures EE fédérées, virtualisation + multi-coeurs Présentation des différents risques d'interférence multicoeur => Évolutivité (Adaptive AUTOSAR, car as a service)

0.2.2 Contraintes et limitations

Difficulté de transition Complexité Coûts

0.3 Objectif(s), contribution et Problématique

Transition partie I - enjeux des fautes temporelles à cause des tendances

Chapitre 1

Enjeux des systèmes à criticité multiple sur processeurs multi-coeurs

Sommaire

L'évolution des système cyber-physiques a progressivement déplacé la complexité des systèmes vers les aspects logiciels, au sein d'une architecture Électrique et Électronique (AEE) de plus en plus complexe. C'est ainsi que l'automobile est successivement passées du tout mécanique vers l'électrification

1.1 Présentation des architectures Hardware

1.1.1 Mono/Multi/Many Cores et GPU

1.1.2 Architectures mémoires, cas des multi-coeurs

1.2 Risques d'interférences

1.3 criticité mixte & contraintes temporelles

1.3.1 Enjeux des usages des multicoeurs avec contraintes temporelles

1.3.2 Problématique : criticité multiple : comment optimiser l'usage des ressources avec garanties temporelles

transition - présentation contenu Etat de l'Art

Chapitre 2

Etat de l'Art

Sommaire

2.1 Optimisation des ressources CPU

2.1.1 Allocation des tâches - optimisation d'ordonnancement

2.1.1.1 Fair scheduling - OS General Purpose

2.1.1.2 Systèmes à criticité mixte

ordonnancements statiques partitionnements spatio-temporels exemple Hyperviseur PikeOS Automotive case : AUTOSAR timing constraints Avionic Case : ARINC653

2.1.2 Autres considérations

en général on va rarement optimiser à 100%, mais du coup exploiter d'autres critères comme la consommation d'énergie, la chauffe etc... Des critères d'isolation des tâches pour des raison de sécurité peuvent aussi être faits... on abordera pas plus que cela ces éléments dans cette thèse.

2.1.3 Limitations et systèmes plus réalistes

2.1.3.1 Limitations des solutions actuelles

2.1.3.2 Systèmes à modes dégradés et améliorations

Disponibilité des tâches à criticité basse. Diminution de priorités. Élongation des tâches à criticité basses et contrôle de budget. sur la mémoire, [Blin 2017] typiquement sur l'ordonnancement Migration de tâches.

Chapitre 3

Principe et architecture pour la gestion de fautes temporelles

Sommaire

In this section we describe our Monitoring and Control Agent (MCA) as a safety mechanism designed to avoid temporal faults in mixed-criticality systems. Its goal is to guarantee critical end-to-end task chain response times by avoiding interferences that could lead to such temporal fault.

The MCA role is first to monitor the state of a HI-criticality task chain to detect potential deadline miss. If such a potential fault is anticipated, then the MCA switches the system to HI-criticality mode, pausing all non essential workload (LO-criticality tasks), to prevent further interference on the HI-criticality tasks and allow a safe termination. To be efficient, the switch must be triggered only when necessary (as a "mode switch procrastination", as called in [Hu 2019]). That is why we also focus on end-to-end deadline, rather than individual task deadlines, in order to avoid false-positive switching, meaning switching to HI-criticality mode although there is slack in the task chain. Indeed, with an end-to-end perspective, we can use the slack given by a task finishing early to compensate the lateness of an other task in the chain.

In the following, we introduce the execution model considered in our work, then we describe the proposed MCA architecture to finally present in more details the principle of the anticipation mechanism.

3.1 Un modèle basé sur des chaînes de tâches pour garantir les contraintes temporelles

Afin d'étudier et développer notre mécanisme de gestion de fautes temporelles dans le cadre d'un système à criticité mixte, nous avons besoin de formaliser la façon de représenter les tâches qui seront à l'étude et leur modèle. Il est à noté que le modèle ici proposé est relativement arbitraire, et choisi essentiellement pour des raisons de commodité. De fait, on retiendra deux critères principaux pour guider le choix de notre modèle : la simplicité d'implémentation et l'accessibilité à des suites logicielles qui peuvent servir de tâches pour simuler un système réel lors de nos test. L'objectif est ainsi de trouver un juste milieu entre un modèle représentatif d'une réalité technique dans les milieux industriels d'une part, et un modèle qui nous évite des sur-coûts de développement pour obtenir une première preuve de concept fonctionnelle.

Ce modèle doit décrire d'une part la méthode d'exécution des tâches, la façon d'interagir entre-elles, notamment pour les tâches à haut niveau de criticité qui sont reliées sous la forme d'une chaîne pour réaliser une fonction critique. Il est à noter que le mécanisme de sûreté de fonctionnement que nous proposons par la suite est *in fine* indépendant du modèle de tâche ici proposé. Il conviendra d'adapter au besoin la partie de Contrôle du mécanisme, de façon à ce que son exécution prenne en compte l'état d'exécution du système selon le modèle de tâche utilisé, s'il est différent de celui présenté ici. Typiquement la vérification des contraintes de précédence peut différer. On aura l'occasion d'aborder rapidement ces aspects par la suite, avec quelques exemples de modifications requises suivant des changements de ce modèle de tâche. *A voir...*

3.1.1 Modèle de tâches

Il manque ici la présentation/définition de "système à criticité mixte" ainsi que ce que l'on va nommer tâches critiques"/"non critiques". A voir comment séparer cela de la partie "cas d'application industrielle"....

Le système ici étudié est dit à niveau de criticité dual. Il exécute un set de tâches logicielles (dite "charge utile") exécutées sur un support logiciel (classiquement, le système d'exploitation). Elles se répartissent entre les tâches à haute criticité d'une part ("tâches critiques"), et à faible criticité d'autre part (non critiques). *c'est la partie à étendre.*

La plupart des hypothèses faites ici se focalisent sur les tâches critiques, tandis que la seule hypothèse forte sur les tâches non critique est la capacité à les stopper (soit un arrêt total soit une mise en pause) et relancer en cours d'exécution. Sous les systèmes type Unix, cela correspond typiquement à l'envoi d'un signal SIGSTOP et SIGCONT. Sans cette condition, les notions de mode nominal et de passage en mode dégradé ne sont pas exploitables pour notre besoin. Chaque tâche critique τ_i est activée et exécutée suivant une période T_i. A chaque période, le *job* $\tau_{i,j}$ correspond à la j^{ieme} exécution de la tâche τ_i. On peut alors noter pour chaque job $\tau_{i,j}$ son moment d'activation $a_{i,j}$, son début d'exécution $s_{i,j}$ et sa terminaison $e_{i,j}$. On considère qu'un job consommes toutes ses données d'entrée (inputs) au début de

son exécution, s'exécute et fourni à la fin de son exécution les données de sortie. Les données d'entrée et de sortie des tâches sont stockées en espace mémoire partagé : la transmission des données d'une tâche à l'autre se fait de façon asynchrone. Cela nous mène à la question de l'interaction entre les tâches et notamment la façon de représenter la précédence.

3.1.2 Modèle de Chaînes de Tâches

Il faudrait commencer par nommer différents modèles d'exécution de tâches existants ici, c.f. [Friese 2018] La question de la dépendance entre les tâches est importante pour aborder le problème des contraintes temps-réel avec une vision plus macroscopique. En effet dans le cadre de l'usage de tâches ayant des contraintes temporelles "molles" (soft real-time), c'est uniquement avec une vision plus globale de l'exécution du système qu'il est possible de tirer au maximum parti des légers dépassements pour éviter dans la globalité d'avoir recours à des politiques d'exécution des tâches plus restrictives, et par conséquent qui sous-exploitent la puissance de calcul disponible. Nous considérons ici la dépendance entre les tâches via les données partagées entre ces dernières selon un modèle type producteurs/consommateurs. Les tâches ont des relations de cause à effet et par conséquent, d'un point de vue strictement fonctionnel on peut décrire le système comme étant une accumulation de fonctionnalités réalisées par l'exécution de tâches successives. Cela permet alors d'introduire la notion de contrainte temporelle fonctionnelle, qui décrivent des contraintes d'exécution de chaînes de tâches bout-en-bout.

On représente une dépendance entre tâches sous la forme de chaînes de tâches, suivant le modèle $\tau_1 \rightarrow \tau_2 \ldots \rightarrow \tau_n$. Dans un tel exemple, τ_1 est la **tâche d'entrée** de la chaîne, tandis que τ_n est la **tâche de sortie** de la chaîne. Notons que ce modèle peut être étendu pour supporter des tâches représentées par un Diagramme Orienté Acyclique (Directed Acyclic Graph - DAG) sans difficulté. Nous nous contentons ici de travailler avec des chaînes directes, sans divergences ou convergences dans le graphe. De fait, cet ajout de complexité dans le modèle de chaîne de tâche n'apporte rien sur les résultats et la globalité de la démarche et ne demande, au demeurant, pas de modifications sur la solution proposée.

J'hésite à présenter ça dans "l'autre sens" : présenter un modèle de chaînes de tâches plus complet (avec div/conv etc.) et au final restreindre le modèle à des chaînes linéaires qu'au niveau du cas d'étude (chapitre 5)

Dans ce contexte, on peut donc définir la relation entre une tâche τ_i et son successeur τ_{i+1}. Pour produire la donnée de sortie du job $\tau_{i+1,k}$ de la tâche τ_{i+1}, ce dernier consommes toutes les données d'entrée en attente provenant des jobs $\tau_{i,j}$. Les données en attente étant celles qui n'ont pas été consommé par le job précédent de τ_{i+1}, i.e. $\tau_{i+1,k-1}$. On peut donc écrire que pour un i, k donnés, sont consommés les données de tous les jobs $\tau_{i,j}$ ssi j tel que $e_{i,j} \leq s_{i+1,k}$ et $s_{i,j} > e_{i+1,k-1}$. Autrement dit, un job $\tau_{i,j}$ n'a un effet sur $\tau_{i+1,k}$ si et seulement si ce dernier est le premier job de τ_{i+1} exécuté après la terminaison de $\tau_{i,j}$.

Dans ces conditions là, on nomme $\tau_{i+1,k}$ le **successeur** du job $\tau_{i,j}$. On note $succ()$ la fonction qui permet de trouver le successeur d'un job donné. Par extension, la fonction itérative $succ^{n-1}()$ permet de trouver le job de sortie d'une chaîne de

tâche donnée, selon le job d'entrée. Pour illustrer cela, on peut prendre l'exemple d'une chaîne de trois tâches $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$, tel que représenté en Figure 3.1. On peut noter que une des exécutions de la chaîne de tâche, débutant par $\tau_{1,1}$, donne : $succ^2(\tau_{1,1}) = succ(succ(\tau_{1,1})) = succ(\tau_{2,2}) = \tau_{3,2}$.

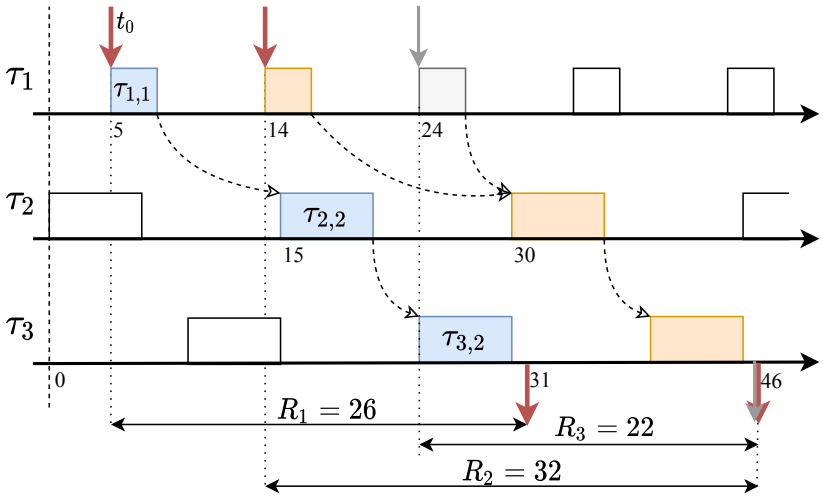


FIGURE 3.1 – Task chain run-time example with $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$

The notion of successor allows the definition of the response time of the chain : it is the elapsed time between the activation of an *entry* task job $\tau_{1,j}$ and the end of the *exit* task job $\tau_{n,k} = succ^{n-1}(\tau_{1,j})$. Noting R_j the response time of this activation, we have $R_j = e_{n,k} - a_{1,j}$. On Figure 3.1 the resulting response times R_1, R_2, R_3 of the first three entry task activation of the chain are represented. Note that with this definition, because tasks can have different periods, several jobs of τ_i can have an effect on a job of τ_{i+1}, as shown in Figure 3.1

Intuitively, an **end-to-end deadline** means that the time it takes for an input of the chain to have an effect on its output, i.e. its response time, must be bounded. Thus, given a deadline D, to be temporally safe our task chain must satisfy : $\max_{j \in \mathbb{N}}\{R_j\} \leq D$.

3.2 Principe de mécanisme d'anticipation - structure Moniteur & Commande

3.2.1 Méthode d'anticipation

Our anticipation mechanism is based on the run-time monitoring of the task chain progress. To that end, we introduce the notions of **Task Chain State** and **Task Chain Execution Trace** (TCET). A TCET contains an entry task job and all the iterative successors of that job. At a time t a TCET can be *active*, if its entry task job has been activated and if its exit task job has not yet ended, or *inactive* otherwise. At time t, the **Task Chain State** is defined as $S(t) = \langle t_0, \tau_i \rangle$ with t_0 the oldest activation among active TCET, and τ_i the next task from this

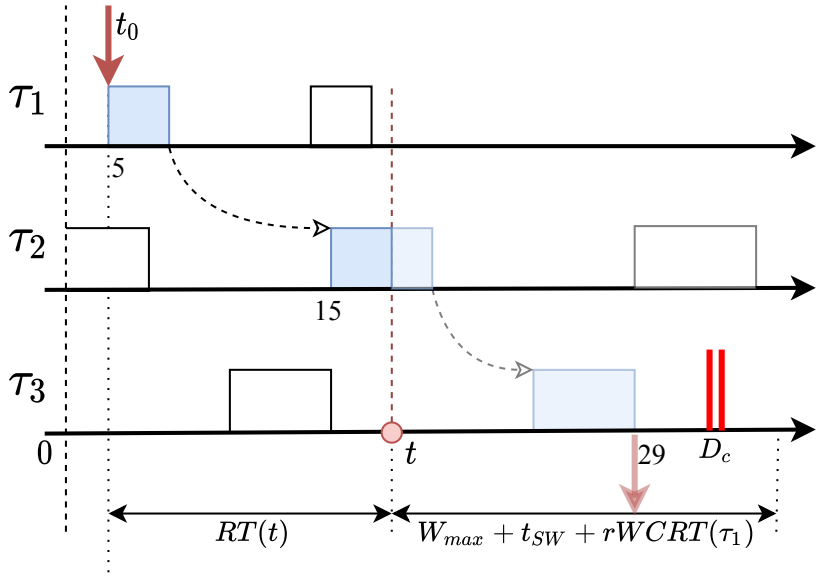


FIGURE 3.2 – active Task Chain Execution Trace with computed anticipation example

TCET to be executed. This way the task chain state indicates the remaining tasks to be executed on the chain and its current response time. Having an estimation of the *remaining Worst Case Response Time* ($rWCRT(\tau_i)$) at that moment for this TCET, the anticipation mechanism can figure out if we are running into a potential deadline miss. For instance, at $t = 18$ on Figure 3.2, the task chain state would be $S(t) = \langle 5, \tau_2 \rangle$. With this $S(t)$ at a given time t, we have the current chain response time $RT(t) = t - t_0$ and the remaining response time $rWCRT(\tau_i)$ estimation (i.e. $\tau_2 \rightarrow \tau_3$ remaining) to check if the execution can finish in time.

Obviously, the estimation of $rWCRT(\tau_i)$ is an important element of the approach. It can be done either experimentally or analytically. We choose an experimental approach for our experiments as the analytical approach is intractable for complex application on a modern multi-core processor or would imply an overly pessimistic estimation. Details of the experimental protocol used for this estimation is given in section **??**. This estimation is made during system integration, without the LO-criticality tasks, thus $rWCRT(\tau_i)$ estimates the worst case time remaining before the end of the task chain if executed in HI-criticality mode.

To decide if it is safe to continue in LO-criticality mode, the anticipation mechanism periodically checks the task chain state. Each observation at a time t is considered temporally safe if the following inequality (adapted from [Kritikakou 2014]) holds :

$$RT(t) + rWCRT(\tau_i) + W_{max} + t_{SW} \leq D \tag{3.1}$$

where W_{max} is the worst time between each observations and t_{SW} the latency to switch to the HI-criticality mode. Let us assume that (3.1) holds, we show that it is safe to wait for the next observation to decide if there is a need to switch. Let t_{next} the time of the next observation. By definition, $t_{next} \leq t + W_{max}$ then necessarily

$RT(t_{next}) \leq RT(t) + W_{max}$, thus
$RT(t_{next}) + rWCRT(\tau_i) + t_{SW} \leq RT(t) + rWCRT(\tau_i) + W_{max} + t_{SW}$. Also, $rWCRT()$ can only decrease as time passes, so $rWCRT(t_{next}) \leq rWCRT(\tau_i)$ and $RT(t_{next}) + rWCRT(t_{next}) + t_{SW} \leq RT(t) + rWCRT(\tau_i) + W_{max} + t_{SW}$. Since (3.1) holds, we have $RT(t_{next}) + rWCRT(t_{next}) + t_{SW} \leq D$.

Hence, it will be safe to switch to LO-criticality mode at the next observation. The setting of the W_{max} parameter is discussed in the next section.

Most of our architectural choices have been made to facilitate portability and deployment of our solution. To that end, the MCA intervenes on the task at the highest level possible and does not require alteration of tasks code or binary.

To help with the estimation of $rWCRT$, we assume that the HI-criticality task chain execute on a single core. To avoid interference between the MCA and the task chain we prevent the MCA to use the same core. Lo-criticality tasks can execute on any core as depicted on Figure 3.3.

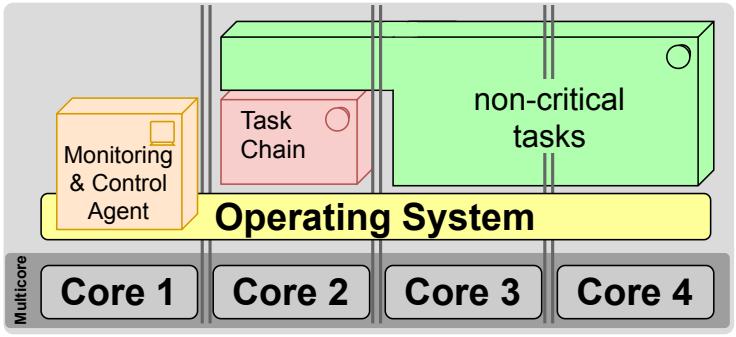


FIGURE 3.3 – Monitoring & Control Agent basic concept

The Monitoring and Control Agent is made of two components : a *Task Wrapper Component* and a *Core Control Component* as shown in Figure 3.4.

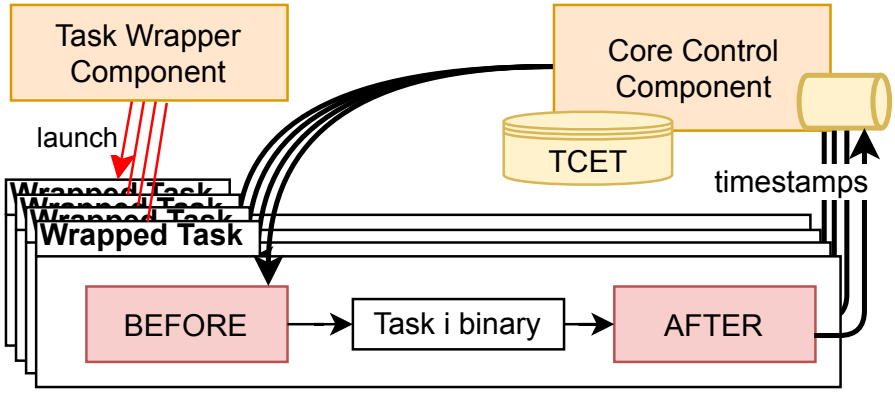


FIGURE 3.4 – Monitoring & Control Agent Architecture

3.2.1.1 Task Wrapper Component (TWC)

It is responsible for encapsulating the system tasks between two software wrappers, “Before" and “After". Those wrappers have two roles :

— provide timestamps (start and end of HI tasks) to the Core Control Component.

— prevent LO tasks execution in HI-criticality mode.

The timestamps are queued to be processed by the Core Control Component to update the TCET. The “Before" wrapper is also used to prevent LO task execution in degraded mode. There is no need for an “After" wrapper for LO tasks.

3.2.1.2 Core Control Component (CCC)

The Core Control Component executes with a period T_{ccc}. It updates each **active Task Chain Execution Trace** (TCET), taking into account timestamps received since its last execution and compute the task chain state $S(t)$, enabling the evaluation of $RT(t)$ and $rWCRT(\tau_i)$. Then CCC checks if inequality (3.1) is still true. If not, the CCC switches to degraded mode to guarantee the task chain deadline. The mode switch is realised through two actions : sending a Pause signal to every LO-criticality tasks, and signaling “Before" wrapper to prevent any new execution.

The CCC parameters t_{sw} and W_{max} are important to define. If those parameters are underestimated, then it is not safe to use inequality (3.1). We estimate them for our experimental platform in **??**. W_{max} is the maximum duration between two CCC checkpoints. It is directly dependent to the CCC period T_{ccc}. If Hi-criticality tasks are periodic, which is typical, it is simple to set this value, around the smallest task period. This way we have the guarantee of not overflowing the timestamps queue used by the CCC. A greater value is possible, but we must take care to process the $TCET$ updates faster than the arrival of timestamps. For other tasks activation models, we must identify the highest task timestamps arrival rate to avoid any queue overflow. It is also important to set T_{ccc} –and thus, W_{max}– as it will directly influence the sensitivity of our anticipation mechanism. With a higher CCC update frequency –and consequently a lower W_{max}– we switch to degraded mode later. Also, it will naturally use more computing resources. A higher value triggers sooner and may increase the number of unneeded switches to degraded mode (i.e. false positives).

3.2.2 Mode dégradé et tâches non vitales

3.2.3 Méthode de recouvrement

3.2.4 structure en Moniteur + Commande - Architecture Logicielle

3.3 Application au domaine automobile (diag. fonctionnel, SWC, etc)

3.3.1 Concept Description

Our approach presents a software execution *Monitoring and Control Agent (MCA)* to guarantee end-to-end deadline constraints. We focus on the respect of end-to-end constraints of tasks chains, not individual tasks constraints. The idea behind this is to offer more “flexibility” on tasks scheduling for guaranteeing mandatory task chains constraints if we control only end-to-end constraints instead of every critical task timing constraint. By doing so, we gain "flexibility" as we allow some parts of the chain to be behind time as they can be compensated before the end of the chain without any external action. The MCA monitors at run-time the execution time of critical tasks and anticipate when the end-to-end deadlines may be compromised to stop non-critical tasks when needed in order to avoid such risk. The anticipation is based on the estimation of remaining WCET. Finally, when the critical task chain recovers from the potential risk, the non-critical tasks can resume their execution to get back to a nominal state.

We define a *degraded mode*, opposed to the *nominal mode* of execution. In nominal mode, critical and non-critical tasks are executed normally. In Degraded mode, non-critical tasks are not executed, to prevent further interferences on critical tasks. The degraded mode implies simpler WCET estimations because we eliminate the disturbances from non-critical tasks ; such WCET will be lower than in a nominal mode. It is probably less pessimistic as we eliminate memory interferences, non-critical tasks scheduling and possible common resources (drivers for instance) usage. The main disturbances remaining will be only between the tasks from the chain. Consequently, our anticipation mechanism will be based on reduced estimation of WCET (compared to nominal mode), to activate degraded mode only as a last resort.

To reach degraded mode, MCA role is to pause/stop non-critical tasks execution. This control is triggered by an anticipation algorithm. To be efficient, this algorithm should trigger the control at the latest possible time while guaranteeing real-time end-to-end constraints.

3.3.1.1 Functional Specification

A critical task chain must describe the implementation of a system functionality from its triggering to its consequence. This would stick most of the time with a computing chain going from a sensor measure to an actuator command. First idea would be to stick with safety criticality levels (ASIL D to ASIL A and QM, for

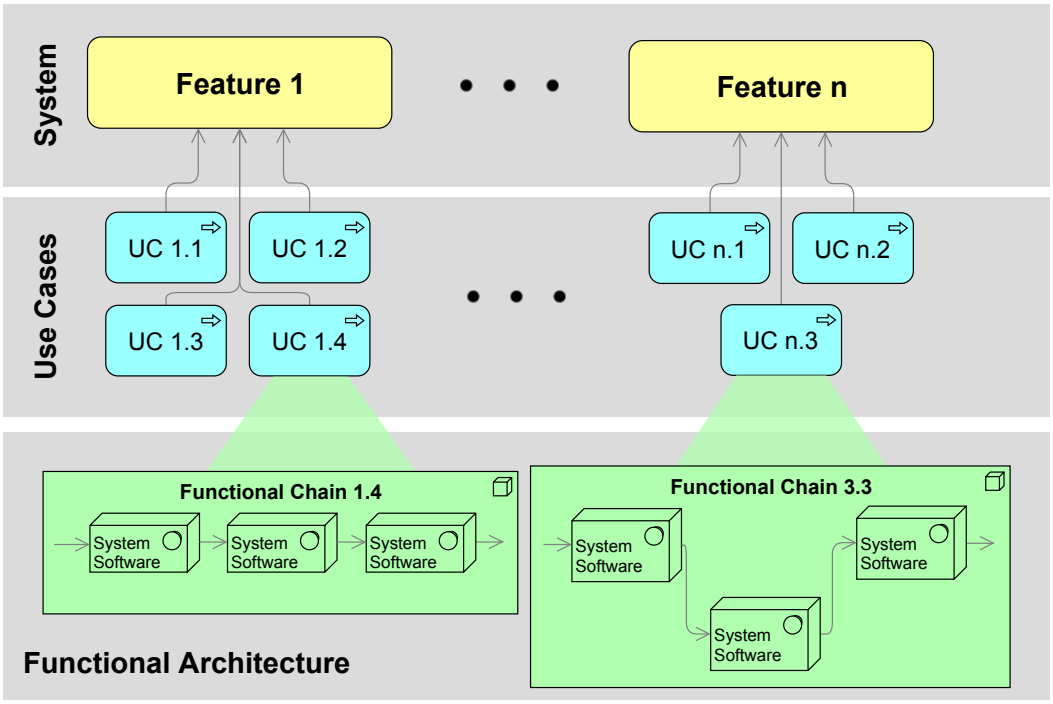


FIGURE 3.5 – Functional Architecture definition

automotive applications), but we quickly notice that there is no direct link between this classification and critical tasks chains. A safety critical task is not necessarily defined from its timing constraints. The only possible conclusion here is that a critical task chain only includes non-QM tasks.

We propose here a definition based around high-level specifications as represented in figure 3.5. The global system is defined as a set of features [1]. Every feature gathers a set of functionalities that are translated into Use Cases [2]. A Use Case defines a feature behavior for a given context and inputs (and the consequent outputs). Finally, those are translated into functional chains representing different functions and their interactions needed for the realization of the Use Case.

If we combine this information with a severity classification in case of failure of the use cases, it is possible to define critical chains as functional chains with a high severity risk. This is one possible criterion allowing an easy separation between a critical functional chain and the others. It could be adapted during the design phase, depending on the functional chains allocated to the processor.

Such information allows to define the software components involved in the critical task chain. All the software components used to realize a critical functional chain form a critical task chain at an OS point of view. At this point, it is possible to define the task chain end-to-end deadline, following the severity temporal risk in case of failure. Such deadline should be at minimum the sum of individual tasks

1. Features : all the services the system must provide. e.g : Lane Support System (LSS) is a feature.
2. e.g : Lane Departure Warning & Lane Keeping Assist are part of the use cases of LSS feature.

deadline, but could probably be higher, depending on the global system and the task chain function. Our objective is to guarantee such critical task chain end-to-end execution time on the multicore.

Chapitre 4

Protocole et démarche expérimentale

Sommaire

4.1 Principe Général et Objectifs

We present in this section the experimental protocol proposed to characterise the system tasks (the "workload") and calibrate the Monitoring and Control agent. The experimental protocol is divided in 7 steps separated in 3 phases : 1. Design phase, 2. Calibration phase, and 3. Run-time validation phase as resumed in Tableau 4.1.

The experimental steps are incremental following two inputs, final step being the complete system with task chain monitoring and control. The first input is the functional load under test that is executed on the real-time framework. It can be either :

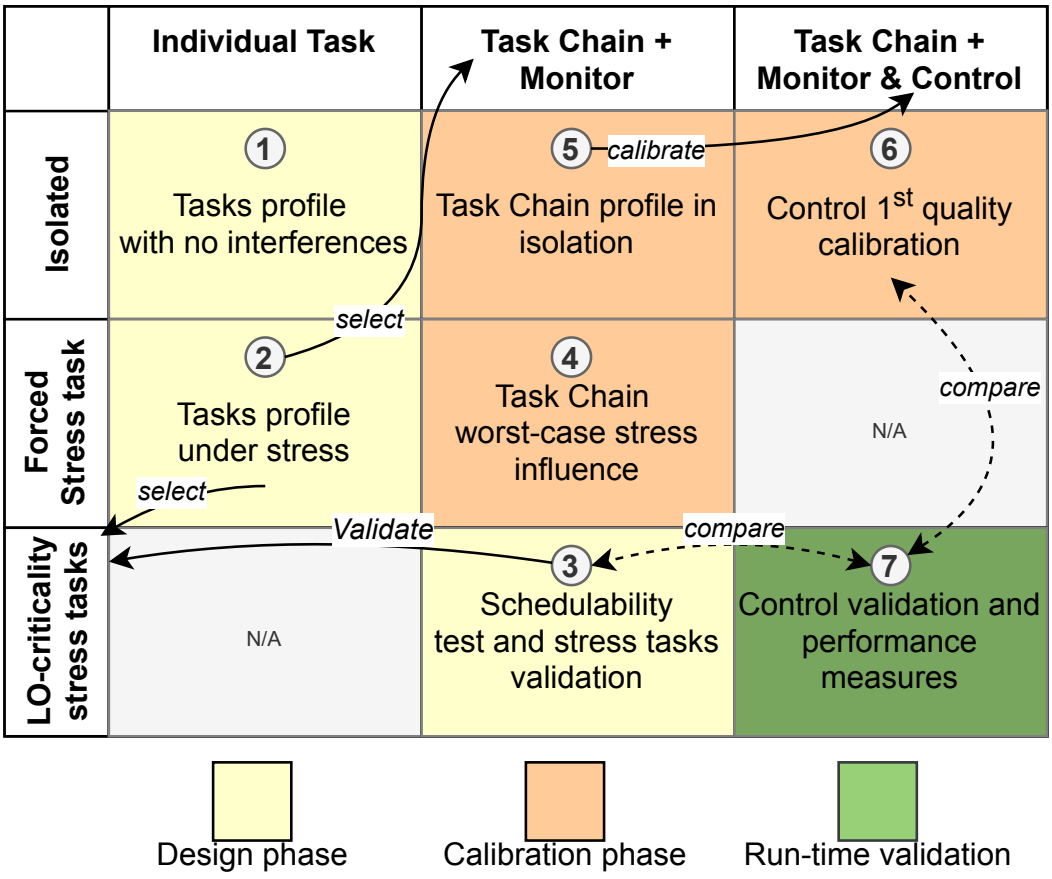
1. single task : a specific task from the workload is executed ;
2. task chain : a specific task chain, made of multiple tasks, is executed and monitored ;
3. Task Chain with Monitoring & Control mechanism.

Second input is the system load executed along with the functional load to influence its execution. It can be either :

1. none, to test an isolated functional load ;
2. forced stress : strong cache, memory and CPU stress from linux `Stress-ng` tool set ;
3. real-time tasks : the LO-criticality tasks of the workload are executed.

Those inputs results in a two entry table as shown in Tableau 4.1 with each box corresponding to a step in the protocol.

TABLE 4.1 – Experimental Flowchart



4.2 Phase de Design

This phase is needed if the workload involved don't come with a detailed specification, including their behavior and execution times. This is the case of our experiments as we select tasks from an already existing benchmark and we have no information about tasks execution times or even their compatibility with our real-time environment. Thus this phase is to characterise the available task set and define the workload specifications. It will be split into the HI-criticality task chain and LO-criticality tasks with their characteristics (min/avg/max execution time, periodicity...). This phase is defined by steps ①, ②, ③ in Tableau 4.1.

4.2.1 Profil des tâches en isolation

First objective is to get a global idea of tasks execution time profiles. One experiment is made per task, the task being executed individually with the framework. The task is called periodically with a given input, and task response times are logged.

4.2.2 Profil des tâches avec stress imposé

We add to the precedent step an artificial system load to cause high stress on cache, memory, I/O and computing use while the tasks are executed one by one. The output is a table with a profile for each task made of the min/average/max execution times and system metrics (system calls, context switches, scheduling interrupts, eventual period misses...). Such profile allow to categorise the tasks following their sensitivity to interferences compared to previous step ①. This allows to define which tasks can be used for the HI-criticality task chain or as stressing LO-criticality tasks but also discard any task that would not fit our needs.

4.2.3 Chaine de tâches avec système complet sans Contrôle

Previous step classified the task set between HI and LO-criticality tasks. We define on this step the specific task chain and LO tasks that will be studied next and verify the pertinence of such choice. We check the workload schedulability in the soft real-time sense (i.e. schedulable if deadlines tardiness are bounded by a reasonably small constant). We also measure the task chain response time profile under "realistic" conditions without the Control mechanism enabled. Expected result is a schedulable system with reference task chain response times with interferences.

4.3 Phase de Calibration

This phase is mandatory to configure the Control mechanism to the software and hardware specificities and lower false-positive rate. It is made of steps ④, ⑤, ⑥ in orange boxes of Tableau 4.1. Configuration includes task chain worst-case response time and intermediary response times in isolation. Performance optimisation consist in tweaking the switch time t_{sw} and anticipation execution frequency W_{max} constants, in the objective of lowering false-positive anticipation rates.

4.3.1 Chaine de tâches avec stress forcé

The task chain is then tested under a worst-case scenario. It is executed with the artificial system load, to stress as much as possible the task chain similarly to step ②. We get a baseline of the worst-case chain response time. This value is important because if the end-to-end deadline is always greater than the worst-case response time observed then the mechanism would be of no use (i.e. deadline never broken from temporal faults). This step gives a quantification of the task chain sensitivity to interferences and thus indicates the pertinence of using a Monitoring and Control Agent to manage them.

4.3.2 Chaine de tâche en isolation

The objective is to calibrate Control mechanism parameters : $rWCRT(\tau_i)$, Core Control Component period (T_{CCC}) and switch time (t_{sw}) to degraded mode. The task chain is executed alone with the MCA but with the Control mode switch disabled. We log every chain intermediary and end-to-end response times. The result gives the data of all the remaining response times obtained during the test. We set the $rWCRT(\tau_i)$ parameters as an upper limits of the remaining response times registered.

4.3.3 Chaine de tâche avec mécanisme de Contrôle

Finally, the Control mechanism is enabled, with the parameters set on previous step. As this step does not include the LO tasks that bring interferences to the task chain, the Core Control Component should not trigger any switch to degraded mode. This step is important for the final analysis as it already points out the base false positive rate obtained with chosen parameters. A qualitative MCA should have the least degraded mode switch possible. Otherwise it could mean that either the CCC parameters are not ideally set (typically W_{max}), or the expected timing delays caused from interferences are too close to the usual timing variation of the task chain execution even in isolation. In other words, the Control Component is not able to differentiate response time variations due to temporal faults from ones due to nominal execution time variations. Another possibility is the end-to-end deadline requirement is too close to the nominal end-to-end response time in isolation.

4.4 Phase de Validation en exécution

4.4.1 Chaine de tâches avec système complet et mécanisme de Contrôle

The validation phase implies a last step (⑦ in green box of Tableau 4.1), which is with the whole final system being executed : HI task chain and LO tasks with the MCA enabled. The objective is to collect the concluding information on the Monitoring and Control Agent behavior to measure the 3 quantification criteria (efficiency, performance and quality) of the solution explained in **??**. We also use the data from steps ③ and ⑥ as a reference for the conclusions.

Chapitre 5

Cas d'implémentation de l'Agent de Monit. & Contrôle

Sommaire

5.1 Framework et Architecture Logicielle

5.1.1 Plateforme Matérielle

The platform used for the experimentation is a barebone computer equipped with a processor Intel Core i5-8250U. This processor embeds 4 cores. It has 3 caches level, L1, L2 and L3 (shared), with respectively 32 KiB/core, 256 KiB/core and 8 Mib (shared). We fixed its frequency to 1400MHz and disabled hyper-threading for our tests.

5.1.2 Support Logiciel

We used Linux (Linux Mint xfce 18.04 distribution) to mix general purpose and real-time applications with different scheduling policies ([Wong 2008], [Lelli 2011]). Its versatility grants easier compatibility with benchmarking suites. Moreover, by adding Xenomai (v. 3.1) real-time co-kernel [Gerum 2004], it is possible to get closer to real-time applications with latencies lowered from milliseconds down to microseconds. It also grants an API for real-time application development, used for the MCA framework.

Notably, POSIX enables to force tasks execution to dedicated cores and change both priority and scheduling policy. As we are in a controlled context that suppose no malicious behavior, we do not implement mechanisms like memory protection or strong space isolation policies. As stated before, vanilla Linux Kernel is not made for hard real-time application. That is mainly because kernel is not preemptive on most parts of it, this can cause high latency for real-time interrupts, from kernel code execution that could be linked to non-critical applications. Therefore, we add a Xenomai co-kernel to improve latency down to micro-seconds and run our MCA

to respect desired real-time constraints. Please note that from Linux point of view, "threads" and "processes" are equivalent and correspond to "tasks" for us.

Threads are assigned 2 parameters, a scheduling policy and a static priority (*sched_priority*). Both are considered by the global scheduler. It first gathers the threads by priority level to execute highest priority processes first. Then for a same priority level, the scheduling policy of each task will define which one to run first. For normal processes the priority level is ignored (considered at 0) to be executed following the CFS policy. This way, a real-time process with a priority level from 1 (lowest) to 99 (highest), always run before them. The threads' scheduling policy defines how they are inserted into the list of same priority level and how they move in this list, all processes being preemptive. We can list 3 real-time policies for real-time process : FIFO, EDF and Round-Robin.
For this purpose, Linux allows to bound threads to cores. For a processor with j cores, every thread has a core affinity represented by an array of j Booleans. Each of these Booleans of affinity b_{Ti} indicates if the thread T can be executed on the core i. By default, every normal process has a core affinity of $\begin{pmatrix}1 & 1 & 1 & 1\end{pmatrix}$, for a quad-core processor, meaning that it can be executed by every core. It makes it easier for the scheduler to balance load between every core. But for our case and when it comes to run hard real-time applications, it is interesting to use such affinity. This way, it will be possible to isolate our MCA on an isolated core and bound the benchmark processes to the other threads. Xenomai is a real-time kernel that can be installed as a co-kernel to a classic Linux distribution as presented in deep by [Gerum 2004]. Our framework and experiments are implemented on the real-time APIs proposed by Xenomai 3.1. In such configuration, it adds an interruption pipeline (ADEOS) directly between the hardware and OS low-level software (i.e. Hardware Abstraction Layer, OS Kernel and drivers). This enables to catch all the interrupts and distribute them in priority to Xenomai real-time kernel. Threads executed with Xenomai are executed either in primary or secondary mode. In both cases they are memory-protected from other processes. By default, Xenomai threads starts in primary mode. They get directly access to Xenomai API and are scheduled by its real-time scheduler. A Xenomai thread can however use kernel API with system calls. When it happens, the Xenomai tasks goes temporarily to the Linux scheduler and automatically goes back to Xenomai domain once done. As the priority system used on primary mode is compatible with the secondary one, the Xenomai tasks keep their highest priority status. It makes the mode switch transparent.

All things considered, we mainly use Xenomai to get a significant latency gain (divided by up to 10) for the critical tasks. We can stay on the classic Linux domain for our non-critical tasks.

Such OS configuration allows us to specify a per-task core allocation and priority level. Linux scheduler as explained in [Ishkov 2015] selects tasks first by priority level, (from 1 to 99 for real-time tasks domain). Then for a given priority level, multiple scheduling policies are possible : Global Earliest Deadline First, FIFO, Round-Robin, and other best-effort policies. To test a system using classic Round-Robin for instance, every task are launched at same priority level with Round-Robin

TABLE 5.1 – MiBench selected tasks

Automotive	basicmath, bitcount, qsort, susan (smooth/edges/corners)
Network	dijkstra, patricia
Consumer	jpeg (code & decode), typeset
Office	stringsearch
Security	blowfish, rijndael, sha
Telecom	adpcm (coding & decoding), CRC32, FFT, gsm

policy. We use Rate-Monotonic scheduling policy for our tests this way.

5.2 Benchmark MiBench

5.2.1 Présentation

MiBench [Guthaus 2001] plays the role of the task set to constitute our experimental workload. This benchmark suite gives source code for 30+ standalone binaries classified in six domains : automotive, security, network, telecommunication, office and consumer. Those tasks do different jobs similar to ones in these domains, with different levels of complexity that is of high interest for us.

To run an artificial system load as a “worst-case" cache, memory, CPU use and I/O stress, we use Linux *Stress-ng* tool presented in [King 2019].

As we do not have yet real industrial application for testing, for now the MiBench Benchmark suite [Guthaus 2001] has been used for our experiments. The objective is to use applications similar as much as possible to computation profiles that could be found in real applications, in order to reproduce memory containment and resource usage close to real cases.

MiBench consists of a large panel of tasks with different memory needs and execution profiles to mimic existing applications. We have at disposal applications from 5 different domains, as presented in the Tableau 5.1. It is used here to validate the framework and put into practice our experiments.

We selected a set of 16 applications from MiBench for our experiments. Most of them exists in “small" and “large" version that allows to change proportionally their execution time and resource needs. Also, some of these tasks may have several variants according to setup parameters. For instance, *Sunsan* has 6 different variants : edge detection, corner detection and smoothing, all 3 existing in both “small" and “large" version which works with a bigger image for processing. This way, those 16 applications leads to 45 different possible tasks for our experiments. It enables to test different combination following the “size" and number of tasks but also the kind of tasks we use. Tasks profile classification were already made by Guthaus & al. in [Guthaus 2001] and detailed work about their memory consumption can be found in [Blin 2016].

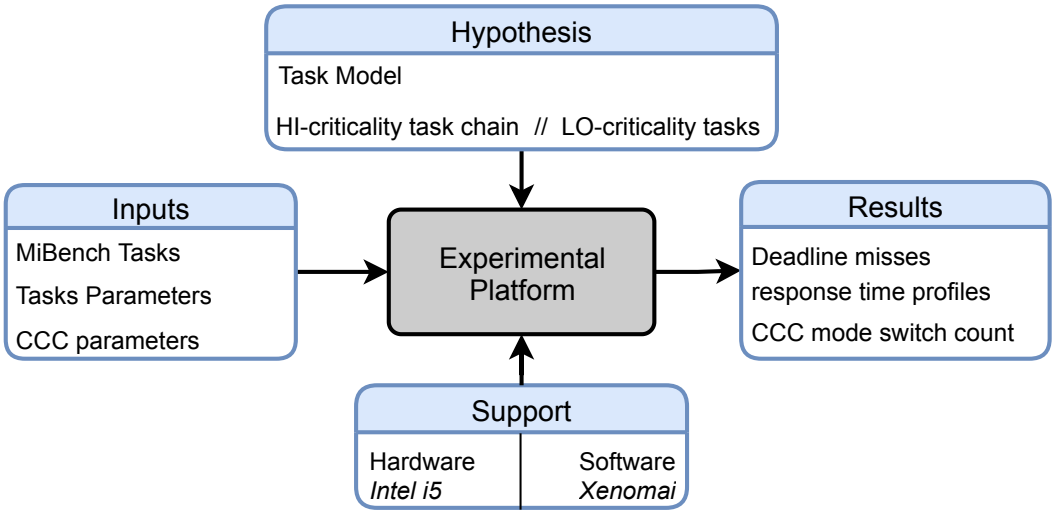


FIGURE 5.1 – Experimental Platform structure

5.2.2 Demandes d'adaptation/modification des tâches

5.3 Agent de Monitoring et Control

5.4 Solutions adoptées à la complexité d'implémentation

Difficultés rencontrées dans la mise en place de ce concept et leçons apprises (en cas de volonté de reproduction)

Chapitre 6

Mise en Application expérimentale

Sommaire

6.1 Application à MiBench du Protocole

Using Mibench as a workload had advantages but also drawbacks. It allows to get specific tasks with a defined and already studied behavior but we are dependent on the way they are initially programmed. They might not completely fit our needs to simulate embedded applications or have incompatibilities with the chosen real-time environment. First step in using this benchmark is to check those criteria to select precisely the tasks from MiBench we use.

6.1.1 Phase de Design

6.1.1.1 Profil des tâches en isolation

We need to establish the execution time profile of each task of the bench. As a result some tasks will be removed from the tests, either due to execution time magnitude differences or inconsistent behaviors between experiments. Accordingly, we measure on each experiment the min, max and median execution times, but also some system counters as the Xenomai mode switches and the amount of linux system calls. Without interferences, the execution time characteristics should have low variations.We see in Tableau 6.1 a sample of the tasks characteristics collected, for 3 different profiles.

With such data, we identified the majority execution time range in MiBench task set around 10ms (from 2-3ms to 20-30 ms) and the basic system calls and mode switch amounts due to initialisation phase (respectively 58 mode switches and ≈hundreds of system calls).

Consequently, we discard tasks out of the execution time magnitude like *adpcmCaudio_L* with an average execution time of 432 ms. By the end of step ①, we retained 34 tasks : Bitcount_L, Bitcount_S, Basicmath_S,

TABLE 6.1 – Tasks profiles in *Xenomai* environment

Task	execution times (ms)		System Counters	
	Median	Max	Mode Switch	Sys. Call
Patricia	0.026	0.099	10051	10338
FFT	7.36	7.39	58	2343
rijndaelE	140,11	141.81	158	446

Basicmath_L, Dijkstra_L, Dijkstra_S, Fft_inv_L, Fft_inv_S, Fft_L, Fft_S, GsmToast_L, GsmToast_S, GsmUToast_L, GsmUToast_S, RijndaelE_S, RijndaelD_S, Sha_L, Sha_S, Stringsearch_L, Stringsearch_S, AdpcmCaudio_L, AdpcmCaudio_S, AdpcmDaudio_L, AdpcmDaudio_S, Cjpeg_L, Cjpeg_S, Djpeg_L, Djpeg_S, Susan_L_corners, Susan_S_corners, Susan_L_edges, Susan_S_edges, Susan_L_smooth, Susan_S_smooth.

6.1.1.2 Profil des tâches avec stress imposé

We add stress on cache level and communication bus from previous step experiments. The objective is to discriminate our tasks in two groups depending on their reaction under stress. If it increases execution time too significantly (more than x10 from average time in isolation) it means the tested task is not suited for the tested environment and suffers not only from interferences but also from LO-criticality tasks preemption. A significant increase in mode switches also indicates such behavior. The tasks that do not pass correctly this test will be either ignored or used LO-criticality stress tasks. Tasks without an exploding execution time or huge increase of mode switches will be used to generate the HI-criticality task chain. Execution time profiles of task used for this purpose are in Tableau 6.2. We finally retained 22 tasks at the end of step ②.

TABLE 6.2 – Tasks profiles in *Xenomai* environment

Task	execution times isolated		execution times stressed	
	Median (ms)	Max (ms)	Median (ms)	Max (ms)
djpeg	1.97	2.28	19.91	211.53
rjindaelD	8.80	9.77	35.02	526.33
FFT	1.85	1.86	2.03	14.8
FFT^{-1}	3.56	3.57	4.05	19.74
bitcount	8.36	9.52	9.98	45.18

6.1.1.3 Chaine de tâches avec système complet sans Contrôle

At this point, we defined our task set, composed of the LO-criticality tasks used as “real" stress and the task chain made of 5 tasks :

$$FFT \rightarrow Bitcount \rightarrow Basicmath \rightarrow FFT^{-1} \rightarrow sha.$$

We need to verify the validity of our choice in term of schedulability and effectiveness of the LO-criticality tasks as interferences. Executing the whole task set together allow to verify both for this step ③.

The right part (blue) of **??** shows the task chain response time distribution profile with the full workload executed (i.e. LO-criticality tasks included). We see the perturbation due to the LO tasks on the critical task chain execution. Our workload is schedulable (no execution drops and deadline misses have reasonable overheads) and the task chain meets high response times compared to its average "nominal" response time for $\approx 10\%$ of the executions (above 200ms response time). We arbitrarily define the task chain deadline $D = 160$ms.

6.1.2 Phase de Calibration

This phase is dedicated to configure the Core Control Component parameters ($rWCRT_i(\tau_i)$, t_{sw} and W_{max}) and run the reference experiments of the task chain behavior on a worst-case stress context (step ④).

6.1.2.1 Chaine de tâches avec stress forcé

In this part we use *Stress-ng* to simulate a worst case stress condition. The task chain potential worst case response time in this context raises at 300ms. Such increase by 100% of the max chain response time under this scenario indicates the pertinence of using a MCA. Regarding such result, our workload stresses the task chain in a significant magnitude.

6.1.2.2 Chaine de tâche en isolation

For step ⑤, we execute the task chain in isolation (i.e. degraded mode). Execution time profile is on the left part (blue) of **??**. We calibrate the Monitor & Control mechanism parameters. We need the different $rWCRT$s for each value of τ_i as defined in **??**. For such linear 5-task chain we logically have $i \in \{1, 5\}$. At run-time, the remaining response times are logged in degraded mode, i.e. the task chain in isolation, and we keep an upper value of the worst measured remaining response time for each τ_i as its $rWCRT(\tau_i)$ in Tableau 6.3. Finally, regarding previous results from step ③, we set $W_{max} = 1$ms, and $t_{sw} = 500\mu$s for our platform.

TABLE 6.3 – Task Chain $rWCRT(\tau_i)$ values in degraded mode

$rWCRT$	τ_0	τ_1	τ_2	τ_3	τ_4
time (ms)	129	93	68	49.5	25

6.1.2.3 Chaine de tâche avec mécanisme de Contrôle

With the previous calibration, we can execute the task chain alone with the Control mechanism enabled. In this isolation case, we should see almost no switch

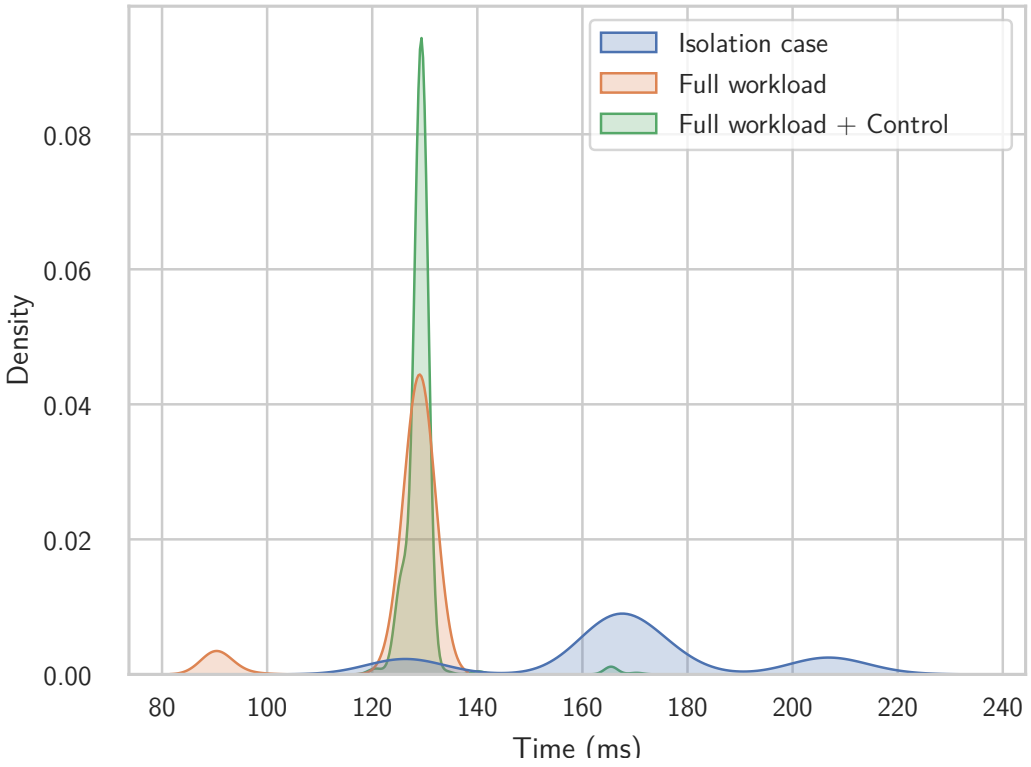


FIGURE 6.1 – Task Chain response time profile from steps ③, ⑥, ⑦

to degraded mode (and on a perfect case, no switches at all) as they must be false-positive. This experiment allows to validate the parameters set on the previous step. On our tests, we measured 0.3% of false positive triggers to degraded mode. The task chain in degraded mode response time distribution profile is illustrated in Figure 6.1.

6.1.3 Phase de Validation en exécution

6.1.3.1 Chaine de tâches avec système complet et mécanisme de Contrôle

As a final experiment, we test the complete workload (HI and LO tasks) with the Monitoring & Control Agent enabled and configured from previous step. First we observe the MCA CPU use, that is inferior to 1%. For a 120s long experiment, it ran for 1.3s overall (including setup time). We were not able to find any difference regarding CPU percentage use with and without our mechanism, either with a big task sets (small tasks only, CPU usage around 80% displayed) and with smaller task sets (e.g. only the task chain described above). Such footprint is low enough to include easily such mechanism.

In term of **efficiency**, our MCA prevented every task chain execution over a 170ms response time. Only 6 occurrences (0.1%) missed the deadline set at 160ms.

The MCA brought down the average response time of the chain from 168ms (no Control enabled) to 129ms. Such value is way closer to the average task response time profile in isolation (125ms). The few missed deadlines can be explained by the implementation framework we used, with a workload (MiBench tasks) not fully compliant with real-time programming constructs recommendations that causes uncontrolled linux system calls for instance. In conjunction with the exacting deadline we arbitrarily set at 160ms while the general workload is demanding (generating 84% deadline misses without the MCA in step ③), this explains this non-perfect result. We could use more pessimistic $rWCRT(\tau_i)$ values to achieve no deadline misses, at the expense of a worse result on the quality criteria. By the end it is a question of compromise, depending on the specific needs.

The **quality** of our calibration seems promising as there were less switches to degraded mode with the Control enabled than the number of deadline misses with no Control at all. This implies that preventing a deadline miss had a more general impact reducing the overall number of timing faults.

In term of **performance**, the system maintained LO-criticality mode for 82s / 120s total, i.e. a performance factor of 0.69 for a loss of 31% of the time in degraded mode.

All those metrics are promising for the use of a Monitoring and Control Agent in order to change a chain response time at an optimum value to avoid the great majority of the deadline misses and on the same time still take few compromises on the LO-criticality tasks execution.

6.2 Conclusions expérimentales

Conclusion et Perspectives

6.3 Conclusion

6.4 Perspectives et améliorations possibles

6.4.1 Mode dégradé multi-niveau

6.4.2 mode dégradé par mécanismes de contrôle hardware

Annexe A

Exemple d'annexe

A.1 Exemple d'annexe

Bibliographie

[Blin 2016] Antoine Blin, Cédric Courtaud, Julien Sopena, Julia Lawall et Gilles Muller. *Understanding the Memory Consumption of the MiBench Embedded Benchmark*. Dans International Conference on Networked Systems, pages 71–86, Marakech, Morocco, 2016. (Non cité.)

[Blin 2017] Antoine Blin. *Vers une utilisation efficace des processeurs multi-coeurs dans des systèmes embarqués à criticités multiples*. phdthesis, Université Pierre et Marie Curie - Paris VI, Paris, 2017. (Non cité.)

[Friese 2018] Max J Friese, Thorsten Ehlers et Dirk Nowotka. *Estimating Latencies of Task Sequences in Multi-Core Automotive ECUs*. 2018. (Non cité.)

[Gerum 2004] Philippe Gerum. *Xenomai - Implementing a RTOS emulation framework on GNU/Linux*. Rapport technique, Xenomai, 2004. (Non cité.)

[Guthaus 2001] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge et Richard B Brown. *MiBench : A free, commercially representative embedded benchmark suite*. Dans 4th International Workshop on Workload Characterization, Austin, TX, USA, 2001. IEEE. (Non cité.)

[Hu 2019] Biao Hu, Lothar Thiele, Pengcheng Huang, Kai Huang, Christoph Griesbeck et Alois Knoll. *FFOB : efficient online mode-switch procrastination in mixed-criticality systems*. Real-Time Systems, vol. 55, no. 3, pages 471–513, 2019. (Non cité.)

[Ishkov 2015] Nikita Ishkov. *A complete guide to Linux process scheduling*, 2015. (Non cité.)

[King 2019] Colin Ian King. *stress-ng - A stress-testing Swiss army knife*, 2019. (Non cité.)

[Kritikakou 2014] Angeliki Kritikakou, Claire Pagetti, Olivier Baldellon, Matthieu Roy et Christine Rochange. *Run-Time Control to Increase Task Parallelism In Mixed-Critical Systems*. Dans 26th Euromicro Conference on Real-Time Systems (ECRTS14), pages 119–128. IEEE, 2014. (Non cité.)

[Lelli 2011] Juri Lelli, Giuseppe Lipari, Dario Faggioli et Tommaso Cucinotta. *An efficient and scalable implementation of global EDF in Linux*. 7th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'11), 2011. (Non cité.)

[Wong 2008] Chee Siang Wong, Ian Tan, Rosalind Deena Kumari et Fun Wey. *Towards Achieving Fairness in the Linux Scheduler*. SIGOPS Oper. Syst. Rev., vol. 42, no. 5, pages 34–43, 2008. (Non cité.)

Résumé : resume **Mots clés :** mots, clefs